Homework 8: Q2

Name: Waiwai Kim, waiwaki

1 Part (a): Algorithm Details

The naive $O(n)$ algorithm is straightforward, as it is stated in the recitation note. The algorithm iteratively multiples a $n$times.

Figure 1: O(n) algorithm

```
4 def pow(a, n):
5     b= 1
6     for i in range(n):
7         b = b*a
8 
9     return  b
```

2 Part (a): Runtime Analysis

Let's assume that a multiplication operaion takes $O(1)$ time in line 7. Note that the for loop in line 6 runs total $O(n)$ times. Thus, the algorithm runs in $O(n) = O(1) \cdot O(n)$.

3 Part (b): Algorithm Idea

We'll build a divide and conquer algorithm here. Also we will do so recursively. First, note that there is a lot of redundant work being done when we try to compute 2^{10}, for example. We can express 2^{10} as $2^5 \cdot 2^5$. We can learn that once we know 2^5, we can save ourselves computing the rest of exponentiation. Similarly, when n is an odd number such as 11, we can express 2^{11} as $2 \cdot 2^5 \cdot 2^5$. The key idea of our divide and conquer algorithm is to divide the size of n into a subproblem of $\frac{n}{2}$.

4 Part (b): Algorithm Details

First, the base case of the algorithm is when n is 1. In this case, the algorithm simply returns a. Second, line 15 of Figure 2 is the recursive portion of the algorithm. Note that a is still the same, while n decreases to $n//2$. $//$ operator in Python returns the integer number in division. Line 17-18 take care of when n is an odd number. Line 19-20 take care of when n is an even number. Because the previous recursive call of the algorithm returns a subproblem of $\frac{n}{2}$, the algorithm must a return the subproblem to the power of 2.

5 Part (b): Proof of Correctness Idea

We will prove the correctness of the algorithm by induction. Consider the case when n=1. In this case, the algorithm returns a. For any given number a, the correct answer is a when $n = 1$.

Figure 2: O(logn) algorithm

```
11 def divide_conquer_pow(a, n):
12     if n==1:
13         return a
14 
15     power = divide_conquer_pow(a, n//2)
16 
17     if n%2 ==1:
18         return a*power*power
19     else:
20         return power*power
```

Next, assume that the divide and conquer algorithm returns a correct answer for all $n \leq k$ where $k > 1$.

Now, show that the algorithm returns a correct answer when $n = k+1$. When $k+1$ is an even number or $k+1 = 2m$ for some integer m. By inductive hypothesis, we know the algorithm of divide_conquer_pow(a,m) returns a correct answer of a^m. Thus, divide_conquer_pow(a,k+1) returns the correct answer because $a^{k+1} = a^{2m} = a^m \cdot a^m$.

When $k+1$ is an odd number or $k+1 = 2m+1$. hypothesis, we know the algorithm of divide_conquer_pow(a,m) returns a correct answer of a^m. Thus, divide_conquer_pow(a,k+1) returns the correct answer because $a^{k+1} = a^{2m+1} = a \cdot a^m \cdot a^m$.

6 Part (b): Runtime Analysis

If statement in line 12 takes $O(1)$. If statement in line 17 also takes $O(1)$. $a \cdot power \cdot power$ in line 18 and $power \cdot power$ take $O(1)$ because multiplying two integers is assumed to take $O(1)$ as it's given to the problem. Finally, line 15 takes half of the runtime of the current level because n becomes halved. Thus, we can express the recurrence relation of the algorithm as shown below in Equation in 1.

$$T(n) \leq \begin{cases} c_1, & \text{if n =1.} \\ T(\frac{n}{2}) + c_2 & \text{otherwise.} \end{cases} \tag{1}$$

In Equation 1, the algorithm takes c_1 when it's on the base case. The key is to express $T(n) = T(\frac{n}{2}) + c_2$ in terms of some constant c_2 and get rid of n on the right-hand side of the equation. We can do so as it's shown in Equation 2. We can transform the first line in Equation 2 to the second line because $T(\frac{n}{2}) = T(\frac{n}{4}) + c_2$. We can continue this process until the right-hand side reaches to $T(1)$. The algorithm gets to the base case when $\frac{n}{2^k} = 1$. This means $k = \log_2^{n}$. The last line of $T(\frac{n}{2^k}) + k \cdot c_2 = T(1) + \log_2^n \cdot c_3 = c_1 + \log_2^n \cdot c_3 \leq O(logn)$. To conclude, the algorithm is $O(logn)$.

$$\begin{aligned}
T(n) &= T(\frac{n}{2}) + c_2 \\
&= T(\frac{n}{4}) + 2 \cdot c_2 \\
&= T(\frac{n}{8}) + 3 \cdot c_2 \\
&= T(\frac{n}{16}) + 4 \cdot c_2 \\
&\vdots \\
&= T(\frac{n}{2^k}) + k \cdot c_2
\end{aligned} \tag{2}$$